Appl. No. 10/083,143
Doc. Ref. **AK1**

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 5 : **G06F 9/45, 9/455, 9/30**	**A1**	**(11) International Publication Number:** **WO 91/20031** **(43) International Publication Date:** 26 December 1991 (26.12.91)

(21) International Application Number: PCT/US91/04073

(22) International Filing Date: 10 June 1991 (10.06.91)

(30) Priority data:

537,466	11 June 1990 (11.06.90)	US
571,500	23 August 1990 (23.08.90)	US

(71) Applicant: SUPERCOMPUTER SYSTEMS LIMITED PARTNERSHIP [US/US]; 1414 W. Hamilton Avenue, Eau Claire, WI 54701 (US).

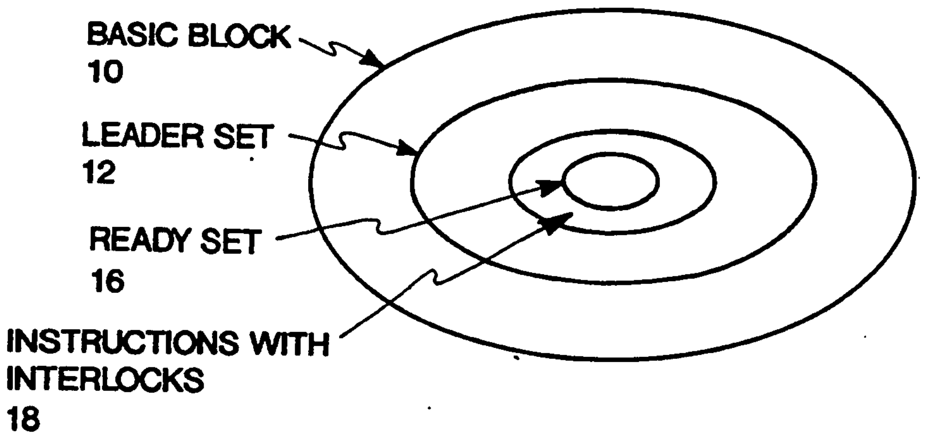
(72) Inventors: RASBOLD, James, C. ; 826 Dana Circle, Livermore, CA 94550 (US). VAN DYKE, Don, A. ; 5133 Independence Drive, Pleasanton, CA 94566 (US).

(74) Agents: PEDERSEN, Brad, D. et al.; Patterson & Keough, 615 Peavey Building, 730 Second Avenue South, Minneapolis, MN 55402 (US).

(81) Designated States: AT (European patent), BE (European patent), CH (European patent), DE (European patent), DK (European patent), ES (European patent), FR (European patent), GB (European patent), GR (European patent), IT (European patent), JP, KR, LU (European patent), NL (European patent), SE (European patent).

Published
With international search report.

(54) Title: METHOD FOR OPTIMIZING INSTRUCTION SCHEDULING



(57) Abstract

A method for scheduling instructions for a processor having multiple functional resources wherein the reordering of the instructions is acomplished in response to a simulation of the run-time environment of the target machine. The simulation of the run-time environment of the target machine is performed at compile time after the machine instructions have been generated by a compiler, or after instruction generation by an assembler. The present invention rearranges the machine instructions for a basic block (10) of instructions into an order that will result in the fastest execution based upon the results of the simulation of the interaction of the multiple functional resources in the target machine.

METHOD FOR OPTIMIZING INSTRUCTION SCHEDULING

TECHNICAL FIELD

The present invention pertains generally to the field of software compiler technology, and to the scheduling of machine instructions to multiple functional resources in a processor. In particular, the present invention pertains to a method of rearranging the order in which machine instructions within a basic block of instructions are issued to a processor containing multiple functional resources so as to reduce the overall execution time of the basic block of instructions.

BACKGROUND ART

In an effort to improve the performance of computer processing systems, present computer processors utilize multiple functional resources for simultaneously performing multiple operations within a single processor. For purposes of the present invention, a functional resource may include any hardware resource directly available to a processor, such as registers, control elements, and, especially, arithmetic and logical functional units. For example, in U.S. Patent No. 4,128,880, the vector processor is provided with three vector functional units and four scalar functional units. The existence of multiple functional units allows this processor to, for example, simultaneously perform two addition operations (one scalar, one vector), thereby increasing the overall performance of the processor. Multiple functional resources are most commonly associated with array or vector processors and scalar/vector processors, but may also be employed in traditional scalar processors.

Another mechanism to improve processor performance that is often associated with high performance processors having multiple functional units is pipelining. Pipelining is a processor implementation technique typically used in vector processors that increases the flow of instructions executing through the processor by simultaneously overlapping the various stages of execution of multiple instructions that each require more than one clock cycle to complete. The work to be done by each multiple-cycle instruction is broken into small pieces such that, at any given time, many instructions are in different stages of execution in the pipeline. Pipelining techniques are also implemented in methods used to feed instructions to the processor and many present art high-speed computers employ instruction pipelines. An instruction pipeline increases the flow of instructions to the processor by maintaining a full queue of waiting instructions that are ready to be fed, or issued, to the processor at the very next clock cycle that the processor can accept the instruction.

While a processor with multiple functional resources has the potential for significantly increased performance, the existence of multiple functional resources greatly increases the complexity of the execution flow of instructions in a processor. For high performance processors, the complexity of execution flow is further compounded by the use of pipelining techniques, both functional unit pipelines and instruction pipelines. For example, consider the execution flow associated with different arithmetic functions that require different amounts of time to complete, e.g., a divide instruction vs. a multiply instruction. In this situation, the different functional unit pipelines will have different latencies. When a processor has multiple functional units with different latencies, it is possible for instruction B to start executing after instruction A has started to execute and complete while instruction A is still executing. Generally, these latencies do not affect the performance of the functional unit pipeline as long as instructions are not dependent upon one another.

Unfortunately, situations known as hazards may prevent the next instruction in the instruction queue from starting during a particular clock cycle. Instruction dependency is a common hazard of this type. For example, an ADD instruction may need to use the result from a currently executing MULTIPLY instruction as an operand. Thus, the ADD instruction is dependent upon the MULTIPLY instruction and cannot execute until the result from the earlier instruction is available. To handle such hazards, the hardware in the processor uses an instruction pipeline interlock that stalls the instruction pipeline at the current instruction until the hazard no longer exists. In the example given here, the instruction pipeline interlock clears when the result of the MULTIPLY instruction is available to the dependent ADD instruction.

One of the ways to optimize the performance of a processor in response to the existence of such hazards is instruction scheduling. Instruction scheduling is a compiler or run-time technique that rearranges the execution order and functional resource allocation of the instructions compiled from a computer program so the instructions execute in the fastest and most efficient order possible. While the rearranged stream of instructions is semantically equivalent to the original stream, the instruction scheduler arranges and overlaps the execution of instructions so as to reduce overall execution time. Instructions are scheduled so as to attempt to minimize any compound impacts their dependency and functional unit latency may have on pipeline performance. The existence of multiple functional resources and the parallelism between mutually independent instructions may also allow the scheduler to hide the latency of the one or more of the processor's functional units and thereby sustain pipelined instruction execution.

Presently, instruction scheduling is typically done using dynamic and/or static scheduling techniques. Some processors, like the CDC 6600 and the IBM 360/91, perform dynamic instruction scheduling as a method of functional resource allocation at execution time that uses a technique called scoreboarding. Scoreboarding is a method of allocating register space that ensures instructions are not issued unless their required register resources are available. Advancements in processor architectures of high performance processors, such as the addition of multiple arithmetic and logical functional units, require scheduling techniques beyond the capability of scoreboarding. For these processors, instruction scheduling must deal not only with functional resources such as registers, but also with the functional unit latencies and other time requirements. Consequently, prior art instruction scheduling may also use static instruction scheduling in an attempt to solve these problems.

Static instruction scheduling is a software-based technique that is done at compile time after machine instructions are generated. Typically, the compiler builds an instruction dependence graph based on the dependencies among the instructions in the instruction stream to be scheduled. Using the instruction dependence graph, the scheduler first generates a preliminary ordering of the instructions in the stream. Next, the compiler estimates the functional unit latency (the time needed for the instruction to normally execute) and the amount of time necessary to accomplish the data transfer from memory for each instruction. Based on information from these two estimates, the scheduler generates a final ordering of instructions.

In prior art instruction schedulers, such as one described by Wei-Chung Hsu in "Register Allocation and Code Scheduling for Load/Store Architectures," it is assumed that all pipeline interlocks are resolved at instruction issue time. This assumption is based on a typical model of instruction issue in which both scalar and vector instructions issue sequentially. In practice, the longer execution times of the vector instructions can clog the issue pipeline and halt the issuance of all instructions. In prior art systems, these halts adversely affect the rate of instruction execution. A new architecture for a scalar/vector processor proposed by the assignee of the present invention addresses this problem by providing for a vector initiation mechanism that is separate from the instruction issue mechanism to prevent the backlog of vector instructions that are halted because of a hardware interlock. This type of processor is not contemplated by prior art instruction schedulers.

Although prior art instruction schedulers are adequate for many pipelined processors, one of the inadequacies of present instruction schedulers is scheduling functional resources for pipelined scalar/vector processors with multiple vector functional units. Such processors have at least two functional units that perform the same set of vector arithmetic operations. In a vector processor with multiple functional units, an instruction may execute in any one of several functional units able to perform the required arithmetic operation(s). This circumstance presents new alternatives that an instruction scheduler must analyze and factor into scheduling decisions. As a result, there is a need for an instruction scheduling method that takes into account that set of information which will enable the instruction scheduler to select the optimum instruction execution path from a set of alternative paths. In addition, there is a need to provide an instruction scheduler that is also takes into account the xistence of alternative models for instruction issue and initiation in a vector processor having multiple functional units.

SUMMARY OF THE INVENTION

The present invention is a method for scheduling instructions for a processor having multiple functional resources wherein the reordering of the instructions is accomplished in response to a simulation of the run-time environment of the target machine. The simulation of the run-time environment of the target machine is performed at compile time after the machine instructions have been generated by the compiler, or after instruction generation by an assembler. The present invention rearranges the machine instructions for a basic block of instructions into an order that will result in the fastest execution based upon the results of the simulation of the interaction of the multiple functional resources in the target machine.

The method of the present invention operates on a basic block of instructions, and first determines which instructions are members of the Leader Set, in a manner similar to prior art static instruction schedulers. The desired issue time (DIT) is then determined for each instruction in the Leader Set, and those instructions which can benefit from early issuance (as determined from the DIT calculation) are moved into the Ready Set. From the Ready Set, instructions are scheduled for issuance in the order of highest cumulative cost.

Leader Set instructions are those instructions that are leading candidates for being scheduled next. An instruction is included in the Leader Set if all the instructions on which it depends have been issued. Because present static instruction schedulers can only estimate the latencies and hardware interlocks among multiple functional resources, the Ready Set in the prior art includes all instructions within the Leader Set that have their dependencies resolved at the time of issue.

Unlike the prior art static instruction schedulers, the present invention determines which instructions will be members of the Ready Set based upon the results of a compile-time simulation. After determining the Leader Set in the usual manner, the present invention then computes a Desired Issue Time (DIT) for each instruction in the Leader Set. The DIT for an instruction is the latest point in time the instruction could issue and still complete execution at the time it would have completed had the instruction been issued immediately. This serves to identify those instructions which can benefit from immediate issuance, and those which could be deferred to a later issue time without a penalty in completion time. Those instructions whose DIT is less than the current value of the simulated time in the compile-time simulation are moved into the Ready Set. This method has the effect of delaying issuance of any instruction that cannot benefit by early execution. By pushing an instruction that cannot benefit from early issue back in the issuance order, the present invention allows instructions that can benefit from early execution to be moved to the Ready Set, i.e., to "bubble" to the top of the instructions to be scheduled. In addition, unlike prior art schedulers, the present invention excludes from the Ready Set those instructions that would encounter a hardware interlock if issued immediately. An instruction whose DIT is greater than the current value of the simulated time is one which will encounter an interlock if issued immediately, and so the method of the present invention treats the instruction as if it is not ready to be issued.

As compared with prior art static scheduling methods, the simulation of actual latencies and hardware interlocks among multiple resources in a processor by the present invention provides a more accurate method for determining which instructions should be in the Ready Set. The accuracy of the membership in the Ready Set is increased, both in terms of those instructions that should have been included in the Ready Set, but were not because of a too conservative estimate of a latency or interlock, and those instruction that should not have been included in the Ready Set, but were because it was not known that they could be issued at a later time and still complete at the same time they would have if they were issued earlier.

The present invention determines the DIT by first predicting the earliest possible issue time (EPIT) for an instruction (based on scalar interlocks), and then simulating the instruction completion time (based on the EPIT just predicted) and comparing completion times on all combinations of available functional resources. The DIT is the latest simulation time that an instruction can issue and still complete by the earliest completion time. In this sense, the DIT is selected as the optimum instruction execution path among a set of alternative paths.

Having determined the members of the Ready Set, the method of the present invention schedules instructions from the Ready Set in an order determined by the relative costs of the instructions. Th "cost" of an instruction represents the cost of not issuing the instruction in terms of how many other instruction depend upon this instruction completing. An instruction upon which many other instructions depend will have a higher cost than an instruction upon which few other instructions depend. Those instructions in the Ready Set with the highest cost are scheduled for execution first. This cost accounting serves to identify and schedule for early execution those instructions critical to execution of the entire basic block.

Although the present invention is applicable to any type of processor having multiple functional resources, the preferred use of the present invention is in connection with a pipelined scalar/vector processor having multiple scalar or vector functional units capable of performing the same arithmetic operations and a vector initiation mechanism that is separate from the instruction issue mechanism. In this type of processor, the vector initiation mechanism is comprised of a secondary vector instruction issue pipeline including a vector instruction queue (5 instructions deep), a vector initiation queue (1 instruction deep), and sets of initiation/dependent initiation control logic by which a waiting vector instruction is either initiated or assigned to execute in a functional unit, or is dependently initiated with vector registers reserved but no immediately available functional unit. In contrast to vector instructions, a scalar instruction in this processor does not issue until all of its operand data is available. Once a scalar instruction issues, its execution (except for memory loads and stores) completes in a fixed number of cycles.

The very highest level of instruction flow and scheduling analysis for the preferred embodiment can be seen as instructions are issued from the instruction issue pipeline. The instruction scheduler can offload numbers of vector instructions into the series of queues contained in the vector initiation pipeline. The vector initiation extension to the instruction issue provides the scheduler with a "launch window" from which to line up the significant vector instruction resource requirements of registers, memory, and functional units. Heavily vectorized portions of code are thus efficiently stacked within the vector processor, thereby decreasing interlock halts in the instruction issue pipeline and the issue of scalar instructions due to an accumulation of waiting vector instructions.

The next level of instruction flow and scheduling analysis for the preferred embodiment can be seen within the vector initiation pipeline. Vector instructions move through the vector initiation pipeline and its interlocks until the vector instructions are released and start execution in a functional unit. For a machine with a vector initiation queue, scalar pipeline interlocks are resolved at issue time, but vector pipeline interlocks are not resolved until after vector initiation time. The scheduler of the present invention uses simulation timings for vector instruction issue, initiation, start, execution, and completion times, and has estimated memory transfer times in ordering the vector instruction stream. Also, the scheduler accounts for dependencies between vector instructions and for scalar/vector instructions having scalar operand dependencies in the scheduled order. Thus, the instructions offloaded into the vector initiation pipeline are scheduled in the order of optimal execution completion times.

Because of the presence of multiple functional units capable of performing the same arithmetic or logic function in the preferred scalar/vector processor, the present invention provides a method for generating path directive instructions that indicate to the processor when the standard rules for the vector initiation should be avoided. When a vector instruction is issued and that instruction can execute on more than one functional unit, the instruction is normally initiated on the functional unit that least recently initiated an instruction. However, based upon the compile-time simulation, the scheduler of the present invention can determine when a vector instruction that would initiate by default on one functional unit in actuality would start earlier on a different functional unit. When the scheduler determines that this is the case, a unique instruction called the PATH instruction is inserted just before the vector instruction being scheduled. The PATH instruction specifies the number of the functional unit to which the subsequent instruction is to initiate, and overrides the default choice made by the control logic in the vector initiation pipeline.

The scheduler of the present invention uses many of the kinds of conventional information available to prior art compiler-based instruction schedulers, including estimates for memory transfer times and instruction dependencies. To optimize the choice of functional units and to determine exactly which functional unit offers the earliest start for a particular instruction, the scheduler of the present invention also considers several new kinds of information. To order the instruction stream, the information considered includes simulation timings for instruction issue, start, execution, and completion times, vector initiation times and the lookahead status of vector functional units. In taking status of the vector functional units, the scheduler looks ahead at the execution stream occurring in all the functional units and, from this look ahead, determines the optimum choice of functional unit for a particular instruction. It is not unusual for this look ahead to include the analysis of 100 to 200 instructions.

Accordingly, it is an object of the present invention to provide a method for optimizing the reordering of the instructions to be executed on a processor having multiple functional resources in response to a simulation of the run-time environment of the target machine.

It is another objective of the present invention to provide an instruction scheduling method that takes into account that set of information which will enable the instruction scheduler to select the optimum instruction execution path from a set of alternative paths among multiple functional units.

It is a further objective of the present invention to provide an instruction scheduler that is takes into account the existence of alternative models for instruction issue and initiation in a vector processor having multiple functional units and is capable of performing effective static instruction scheduling for pipelined processors having an extended vector instruction initiation.

Another object of the present invention is to provide a method for generating path directive instructions that indicate to the processor which arithmetic or logical functional unit should be used to perform a given instruction.

A further objective of the present invention is to increase the rate of instruction flow, and decrease the instruction issue pipeline halts in a scalar/vector processor having multiple functional units.

These and other objectives of the present invention will become apparent with reference to the drawings, the detailed description of the preferred embodiment, and the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1a is a Venn diagram of the Leader Set and Ready Set for a Basic Block as generated by prior art static instruction schedulers.

Fig. 1b is an equivalent Venn Diagram to that shown in Fig. 1a of the Leader Set and Ready Set for a Basic Block as generated by the present invention.

Fig. 2 is an example of a dependency directed acyclic graph used by the instruction scheduler of the present invention.

Fig. 3 is a process diagram showing a typical prior art instruction scheduling method.

Fig. 4 is a process diagram showing the instruction scheduling method of the present invention.

Fig. 5 is a process diagram showing the method for computing the DIT of instructions, and for inserting PATH instructions where appropriate, according to the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Referring now to Figs. 1a and 1b, an overview of the difference between the present invention the prior art methods of generating the Leader Set and Ready Set for an instruction scheduler will be described. Present compilers manipulate source code programs in the process of creating an executable file by identifying the basic blocks of the program. A basic block 10 is defined as a set of instructions having a single entry and a single exit. The Leader Set 12 is a set of instructions in a basic block that are the leading candidates for scheduling next. It consists of all instructions in a basic block where the instructions upon which the instruction in question depends have at least issued. Both the prior art instruction schedulers and the present invention create the Leader Set 12 in a similar manner, as described in greater detail in connection with Figs. 2 and 3. The Ready Set 14, 16 is the set of instructions for a basic block that are determined to be ready to issue at a given point in time. As can be seen, the Ready Set 14, 16 is a subset of the Leader Set 12.

The difference between the prior art and the present invention is in the generation of the Ready Set 14, 16. Although it may not be apparent, a person skilled in the art will appreciate that the objective of any instruction scheduling program is to make the Ready Set 14, 16 as small as possible, thereby increasing the probability that instructions in the Ready Set 14, 16 will, in fact, execute to completion without hardware interlock. Because present static instruction schedulers can only estimate the latencies and hardware interlocks among multiple functional resources, the prior art Ready Set 14 includes all instructions that have their dependencies resolved at the time of issue. Unlike the prior art static instruction schedulers, the present invention determines which instructions will be members of the Ready Set 16 based upon the results of a compile-time simulation.

The present invention tests each member of the Leader Set 12 for inclusion in the Ready Set 16 by first computing a Desired Issue Time (DIT) for each instruction in the Leader Set 12. Those instructions whose DIT is less than the current value of the simulated time in the compile-time simulation are included in the Ready Set 16. This method has the effect of delaying issuance of any instruction that cannot benefit from early execution. The instructions that cannot benefit from immediate issuance are those that would encounter an interlock if executed immediately, and are represented by "Instructions with Interlocks" 18 in Fig. 1b. By pushing these instructions back in the issuance order, the present invention allows instructions that can benefit from immediate execution to be moved to the Ready Set, i.e., to "bubble" to the top of the instructions to be scheduled. In other words, the simulation of actual latencies and hardware interlocks among multiple resources in a processor by the present invention provides a more accurate method for determining which instructions should be in the Ready Set. Although for any given basic block, the number of instructions in the Ready Set at any given time may be different than with the prior art Ready Set, the accuracy of the membership in the Ready Set is increased. Those instructions that should have been included in the Ready Set, but were not because of a too conservative estimate of a latency or interlock, are added to the Ready Set. Those instruction that should not have been included in the Ready Set, but were because it was not known that they could be issued at a later time and still complete at the same time they would have if they were issued earlier, are deleted from the Ready Set. The end result is that the rearrangement of instructions by the scheduler of the present invention will result in a more optimal organization of instructions and, hence, increased performance for the processor.

In determining membership in the Leader Set 12, the instruction scheduler of the present invention first constructs a dependency DAG (directed acyclic graph) for the block of code 10 to be scheduled. Typically, instruction schedulers perform scheduling over a basic block of an intermediate language in which the intermediate language closely approximates the assembler language of the target machine.

In the following discussion, the process of scheduling instructions for execution will be described. The scheduling is performed at compile (or assembly) time, and is done by analyzing a simulation of the run-time environment of the target machine. It is therefore conceptually helpful to think of instructions which have been scheduled as having been "issued" in the simulation sense. Since the method of the present invention concerns code optimization techniques at compile or assembly time, those skilled in the art will recognize that reference to an instruction having been issued actually refers to issuance in the simulation, not a target machine.

Referring now to Fig. 2, source code, intermediate language code, and a corresponding dependancy DAG are shown. Source equations 20 are shown along with their representation in a series of nine intermediate language statements 21 that approximate assembly level instructions. From the intermediate language instructions 21, the scheduler generates a dependency DAG 22. Each node of the dependency DAG 22 corresponds to one assembly language instruction, as correspondingly numbered in Fig. 2. For example, node 22a of the dependency DAG 22 corresponds to assembly language instruction 21a. The edges between the nodes in the DAG 22 are drawn as arrows and indicate dependencies between instructions. The edge 24 between node 22b and node 22g indicates that instruction 21b must execute before instruction 21g. Nodes with no incoming edges have no dependencies in the basic block, and need not wait for any other instruction before they can be issued.

The nodes in the dependency DAG 22 are then each assigned a cumulative cost value in reverse order. Conceptually, the "cost" of an instruction is the time consequence of not issuing the instruction. In other words, an instruction from which many others depend has a higher cost than an instruction from which few others depend. Arithmetically, the cost of an instruction is calculated as the execution time of the instruction plus the cumulative execution times of all other instructions which depend therefrom. Another way of expressing this is to say that the cost of an instruction is the cost of all the successor nodes within the basic block plus the time of execution of the current node. The relative costs calculated in this way from the dependency DAG 22 identify the critical paths of instructions in the basic block. Taken as a whole, the dependency DAG 22 thus imposes a partial ordering on the instructions in the basic block. The scheduler is free to reorder the instructions based on the dependencies, as long as the partial order is maintained.

The process of reordering instructions according to the present invention, occurs through a simulation of the run-time environment of the target machine. Through use of the DAG 22, each instruction in the basic block which has not yet been scheduled is placed in the Leader Set 12 if it is not dependant upon any unissued instructions. A leader instruction is identified in a DAG as a node with no predecessors. For example, in DAG 22, nodes 22a, 22b, 22d, and 22e are leader nodes representing intermediate language instructions 21a, 21b, 21d, and 21e. An instruction is not eligible to be issued until it becomes a leader. As instructions are issued, their nodes are removed from the DAG 22 and certain successor nodes become new leaders. Once promoted to the Leader Set 12, an instruction remains in the Leader Set 12 until it is issued.

Fig. 3 shows a prior art instruction scheduling method. Scheduling is done through manipulation of a DAG created for the basic block being scheduled, as described above. First, the prior art method identifies the Leader Set 12 at step 40 as the instructions corresponding to nodes in the DAG 22 with no incoming edges. Of the instruction in the Leader Set 12, those that have no estimated interlocks are moved into the Ready Set 14 at step 42. Interlocks are caused by machine resources such as registers which are needed by an instruction, but are unavailable, typically due to their being used by a still executing previous instruction. Interlocks are statically estimated based upon a predefined instruction completion time. Instructions in the Leader Set 12 without interlocks are ready to execute, and are thus properly placed in the Ready Set 14, according to prior art methods.

If the Ready Set 14 contains one or more instructions at step 44, the prior art method schedules the instruction in the Ready Set 14 having the highest cost at step 46. The scheduled instruction is then removed from the DAG 22 at step 48, the machine resources are assigned to the scheduled instruction at step 50, and any instructions in the Ready Set 14 which now would encounter an interlock are moved from the Ready Set 14 into the Leader Set 12 at step 52. The process then iterates by returning to the beginning to schedule additional instructions.

In the event that the Ready Set 14 is empty at step 44, the process then checks the Leader Set 12 at step 54 to see if it is also empty. If the Ready Set 14 and the Leader Set 12 are both empty, this indicates that there are no more instructions in the basic block to be scheduled, and the process ends. If the Leader Set 12 is not empty at step 54, then interlocks due to currently executing instructions are cleared at step 56 for the instruction which would be the next to complete. This of course requires that the process make estimates of instruction execution times and instruction completion times, so as to estimate when interlocks caused by executing instructions would clear.

At each iteration of the prior art method shown in Fig. 3, the scheduler chooses the instruction with the highest cumulative cost from the set of instructions which have no pending pipeline interlocks. The prior art is effective because it chooses to issue the instruction without interlocks on the most critical path. Once issued, each instruction will execute to completion.

While the above described method is effective in prior art machines, it is less effective when a vector processor has an extended vector issue pipeline, as is the case in the preferred embodiment, or when the machine has multiple functional units, each capable of executing a given function type. In this circumstance, vector instructions may suffer pipeline interlocks after issue time, since at issue time, a vector instruction has cleared only scalar interlocks, and has been allowed to join the initiation queue. A vector instruction may experience further delays due to vector register access restrictions or functional unit restrictions, or may experience a stall due to an interlock affecting a preceding vector instruction in the initiation queue.

Unlike the prior art scheduling method, scheduling according to the present invention is done through a simulation of the target machine's run-time environment, as well as through manipulation of a DAG 22 created for the basic block being scheduled. The present invention differs from prior art in the method for moving vector instructions to the Ready Set 14 and in the method for tracking the scheduler's simulated time. By performing a thorough simulation of the target machine at the system clock cycle level, the instruction execution times can be accurately predicted rather than merely estimated, and ex cution tim s using alternate execution paths through alternate functional units can be determined, so as to select the most efficient of the available execution path options. Those skilled in the art will recognize that the details of the simulation depend upon the target machine in question, and are thus beyond the scope of the present specification. In the preferred embodiment, simulation is facilitated through use of an instruction description table, which includes information on each instruction's execution time, the control registers implicitely read or written during execution (such as vector mask), and explicit register (such as for operands or results).

Referring now to Fig. 4, a method of scheduling instructions according to the preferred embodiment of the present invention is shown. As in the prior art, the process according to the present invention operates on a basic block of code, and begins by moving all instructions into the Leader Set 12 whose corresponding nodes in the DAG 22 have no incoming edges at step 40. The desired issue time (DIT) for each of the Leader instructions is then calculated at step 60. The DIT for an instruction is the latest point in time at which an instruction can be issued, and still complete at the same time it would have completed had it been issued immediately. For example, if an instruction issued at t=0 would complete at t=200, then the DIT for the instruction is the latest point in time for issuing the instruction, while having it complete execution at t=200. Details of DIT calculation are described in reference to Fig. 5.

Having calculated the DIT for all instructions in the Leader Set 12 at step 60, the process then moves into the Ready Set 16 any instruction whose DIT is less than the current simulation time at step 62. If the Ready Set 16 is empty at step 44 and the Leader Set 12 is empty at step 54, all instructions in the basic block have been scheduled and the process ends. If the Ready Set 16 is not empty at step 44, then the instruction in the Ready Set 16 having the highest cost is scheduled at step 46, and its node and outward edges are removed from the DAG 22 at step 48. The simulation time is then advanced at step 64 to the point in time when the simulation determines that the instruction just scheduled would actually issue. The machine resources such as registers and the functional unit are assigned to the scheduled instruction at step 50, and any new interlocks caused by the assignment of machine resources ar checked to see if instructions in the Ready Set 16 need to be moved back into the Leader Set 12 at step 52. The process then returns to the beginning, to schedule the next instruction.

In the event that the Ready Set 16 is empty at step 44, and the Leader Set 12 is not empty at step 54, the simulation time is then advanced to the point where an instruction in the Leader Set 12 can be moved into the Ready Set 16 at step 66. The process then returns to the beginning, and continues until all instructions in the basic block have been scheduled.

Those skilled in the art will recognize that many modifications can be made to the method described in Fig. 4, without departing from the scope of the present invention. For example, the order of performing steps 48, 64, and 50 is not critical to proper operation of the method.

Referring now to Fig. 5, a method of determining the DIT of an instruction 60 is shown. First, the earliest possible issue time (EPIT) for the instruction is determined at step 70. This will occur when scalar interlocks in the machine have cleared for the instruction in question. If the instruction is a scalar instruction at step 72, then the DIT is set equal to the EPIT at step 74, and the process ends because, if issued at EPIT, the scalar instruction will execute to completion without encountering any interlocks. Delaying issuance of a scalar instruction beyond EPIT will thus delay the completion time of the instruction.

If the instruction in question is a vector instruction (step 72), the simulation predicts the completion time of the instruction at step 76, assuming that the instruction was issued at the EPIT to the default functional unit. In the preferred embodiment of the present invention, the target machine has vector instruction pipeline, and an instruction initiation procedure. The significance of the target machine architecture to the method shown in Fig. 5, is only that the simulation of the target run-time environment must be cognizant of the specific machine architecture involved, in order to accurately predict instruction completion times. So in the preferred embodiment, when a vector instruction is moved to the Leader Set 12, the earliest initiation time of the instruction is calculated based on the EPIT and vector pipeline interlocks that the instruction will encounter in the initiation queue. The DIT is then determined by computing the latest possible issue time of the instruction that will result in initiation at the earliest possible time. The DIT will always be greater than or equal to the EPIT. In the preferred embodiment, an instruction moves from the Leader Set 12 to the Ready Set 16 when the simulated time reaches the DIT.

The process next determines at step 78 whether there are more than one functional units on which the instruction could execute. If not, then the DIT is set at step 80 to be equal to the latest point in time that the instruction could be issued so as to still complete at the predicted completion time. If there are more than one functional units on which the instruction could complete (step 78), the process predicts the completion time for the instruction at step 82, for each of the alternate functional units, again assuming issuance at EPIT. If completion would be earlier on an alternate functional unit (step 84), then the instruction is marked as needing a PATH instruction at step 86, to override the default choice of functional units. When that instruction is scheduled, if an alternate path is preferred, a PATH instruction is inserted into the instruction stream. Either way, the process sets DIT at step 80 according to the earliest predicted completion time, as previously determined.

Although the description of the preferred embodiment has been presented, it is contemplated that various changes could be made without deviating from the spirit of the present invention. Accordingly, it is intended that the scope of the present invention be dictated by the appended claims, rather than by the description of the preferred embodiment.

What is claimed is:

CLAIMS

1. In a computer, a method of optimizing code for a target computer having multiple functional units, comprising the steps of:

   simulating code execution of the target computer;

   reordering code instructions so as to improve execution speed as indicated by the simulation.

2. In a computer, an improved method of scheduling instructions in a basic instruction block for execution, the method comprising the steps of:

   identifying a leader set of instructions within the basic block;

   identifying a ready set as those instructions in the leader set that would execute without interlock interruption if issued immediately; and

   issuing the instruction in the ready set with the highest cumulative pendency cost.

3. In a computer, an improved method of scheduling instructions in a basic instruction block for execution, the method comprising the steps of:

   identifying a leader set of instructions within the basic block;

   computing a completion time for each instruction in the basic block, wherein the completion time for each instruction is based upon immediate issue of the instruction;

   computing a desired issue time for each instruction in the leader set, wherein the desired issue time is the latest point in time at which the instruction can be issued and still complete by the completion time;

   identifying a ready set of instructions as being those instructions in the leader set whose desired issue time is immediately or earlier; and

   issuing the instruction in the ready set with the highest cumulative pendency cost.

4. In a computer, as part of a method for scheduling instructions within a basic instruction block for execution, a method of determining a ready set of instructions comprising the steps of:

   identifying a leader set of instructions as being those without data dependencies from other instructions within the basic instruction block;

predicting instruction completion time for each instruction in the leader set;

computing a desired issue time for each instruction in the leader set, wherein the desired issue time is the latest point in time at which the instruction can be issued and still complete by the predicted completion time;

identifying a ready set of instructions as being those instructions in the leader set whose desired issue time is immediately or earlier; and

issuing the instruction in the ready set with the highest cumulative pendency cost.

5. The method of claim 4 further comprising the steps of:

determining the optimum path of execution among a plurality of available functional units for an instruction; and

if the optimum path of execution is different than a default path of execution, inserting an instruction identifying the optimum path of execution among the plurality of available functional units.

6. In a computer, as part of the method for scheduling instructions within a basic block of instructions, a method for determining which of a plurality of available functional units an instruction will execute on, comprising the steps of:

determining the optimum path of execution among a plurality of available functional units for an instruction; and

if the optimum path of execution is different than a default path of execution, inserting an instruction identifying the optimum path of execution among the plurality of available functional units.

Fig. 1a

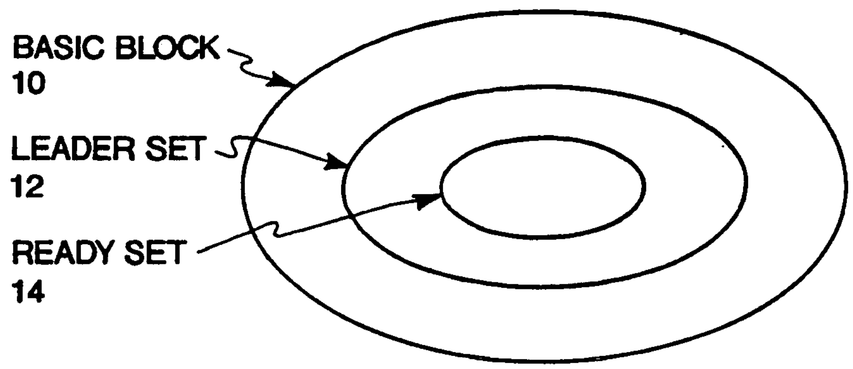

BASIC BLOCK
10
LEADER SET
12
READY SET
14

Fig. 1b

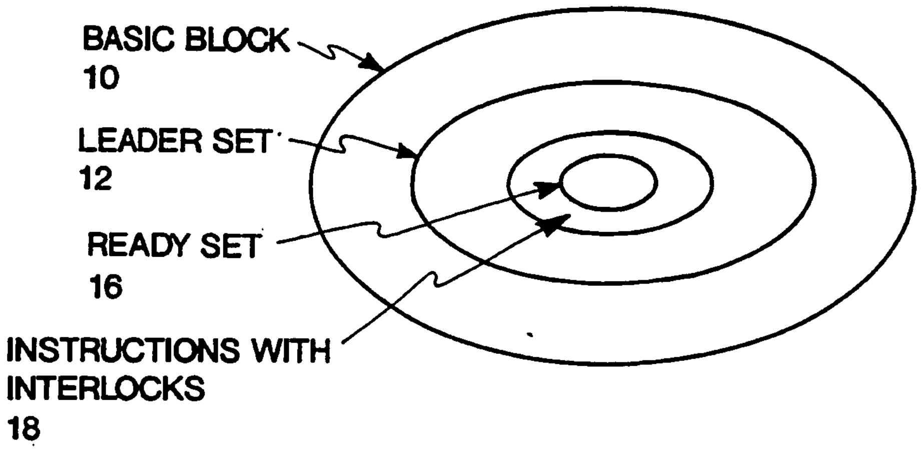

BASIC BLOCK
10
LEADER SET
12
READY SET
16
INSTRUCTIONS WITH
INTERLOCKS
18

Fig. 2

c = a * b
d = b + (e − f)
h = c * d

20

```
21a ──► 1. LOAD  R1,a
21b ──► 2. LOAD  R2,b
21c ──► 3. MULT  R3,R1,R2
21d ──► 4. LOAD  R4,e
21e ──► 5. LOAD  R5,f
21f ──► 6. SUB   R6,R4,R5
21g ──► 7. ADD   R7,R2,R6
21h ──► 8. MULT  R8,R3,R7
21i ──► 9. STORE R8,h
```

21

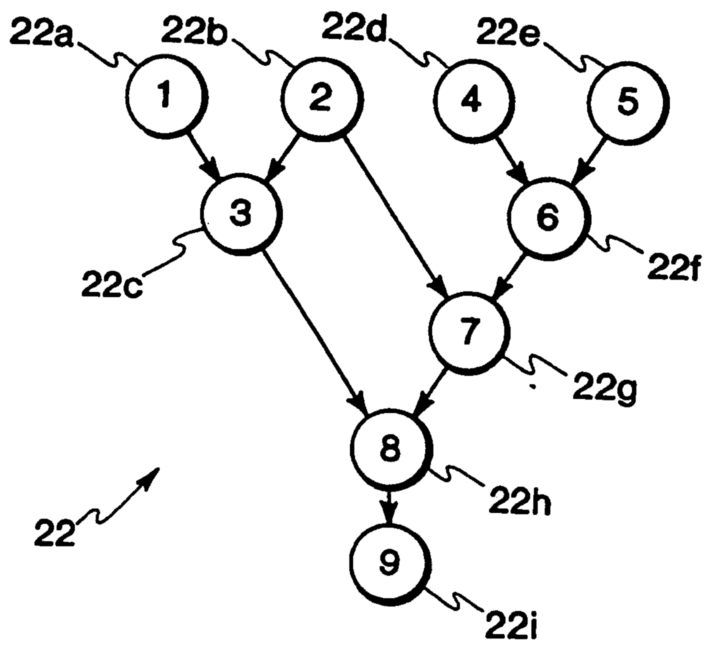


Fig. 3

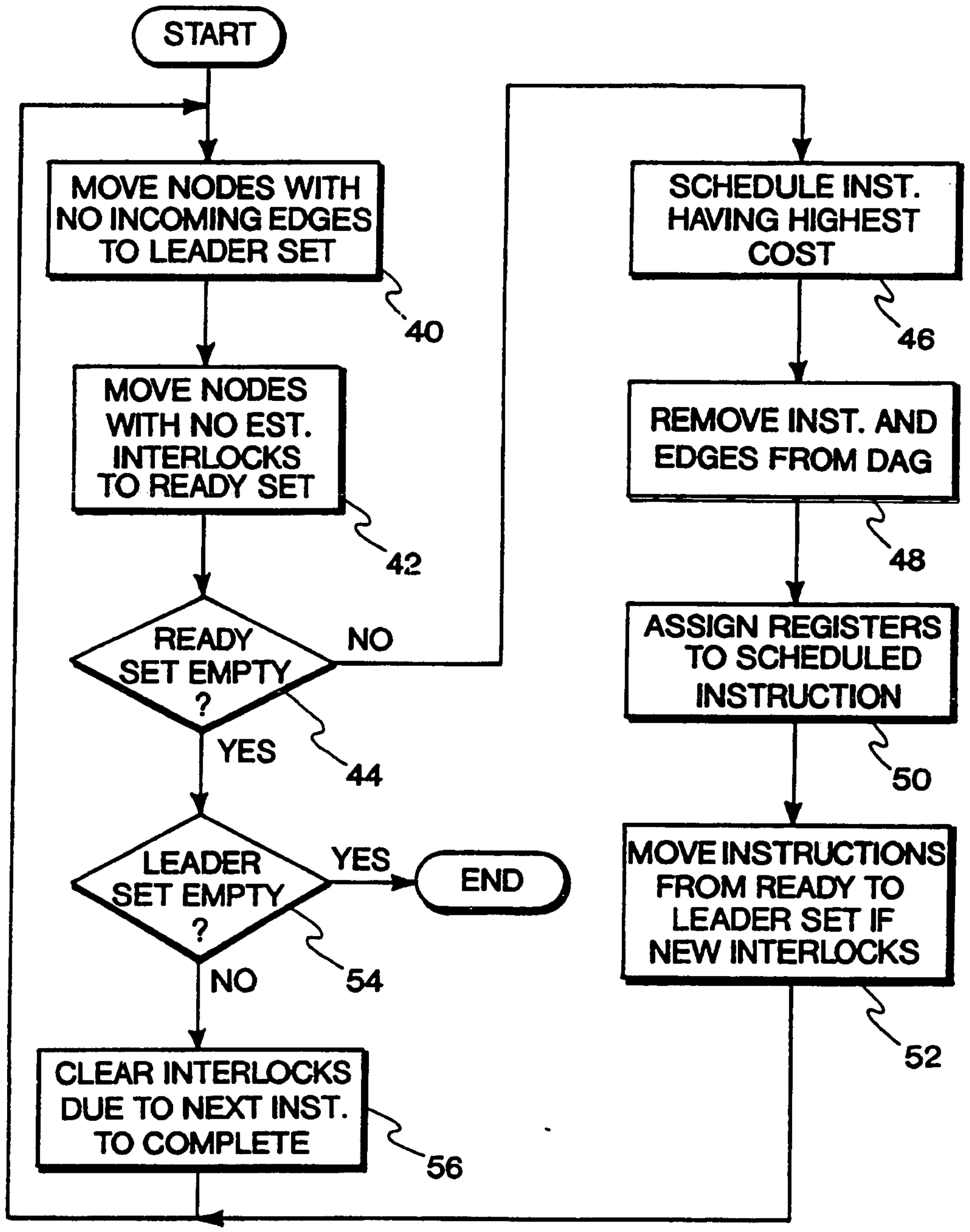

START
MOVE NODES WITH NO INCOMING EDGES TO LEADER SET
40
MOVE NODES WITH NO EST. INTERLOCKS TO READY SET
42
READY SET EMPTY ?
NO
YES
44
LEADER SET EMPTY ?
YES
END
NO
54
CLEAR INTERLOCKS DUE TO NEXT INST. TO COMPLETE
56
SCHEDULE INST. HAVING HIGHEST COST
46
REMOVE INST. AND EDGES FROM DAG
48
ASSIGN REGISTERS TO SCHEDULED INSTRUCTION
50
MOVE INSTRUCTIONS FROM READY TO LEADER SET IF NEW INTERLOCKS
52

Fig. 4

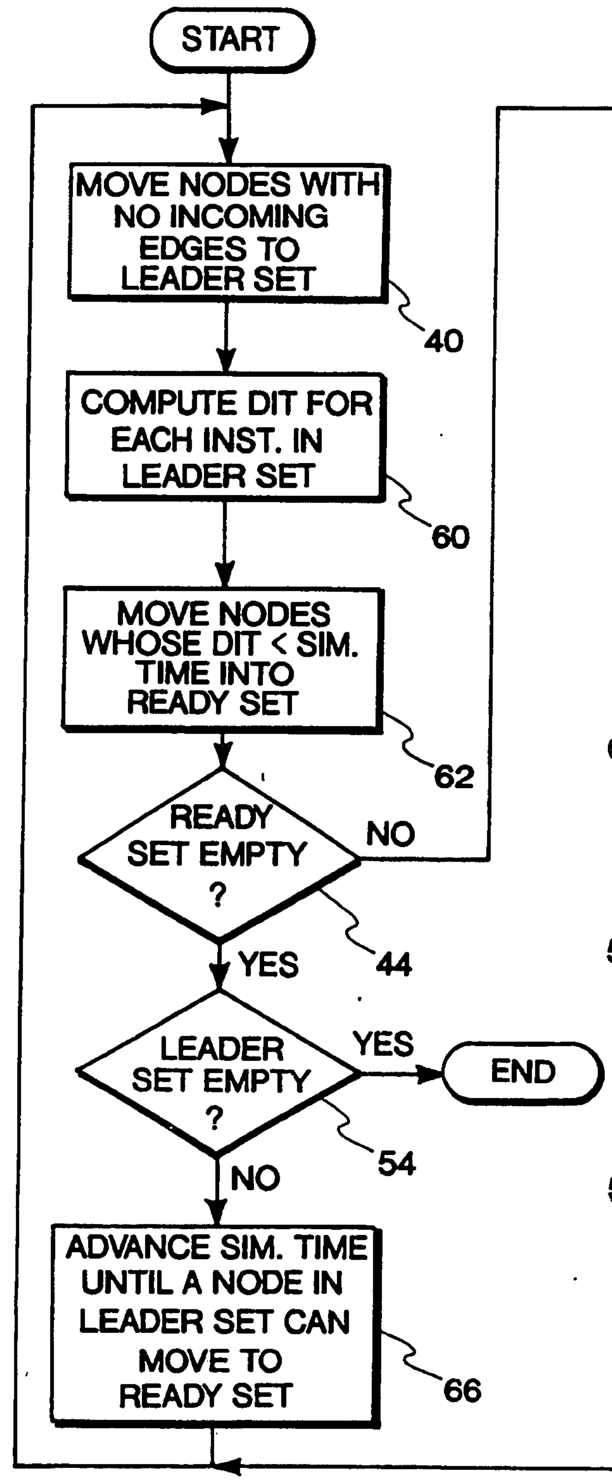
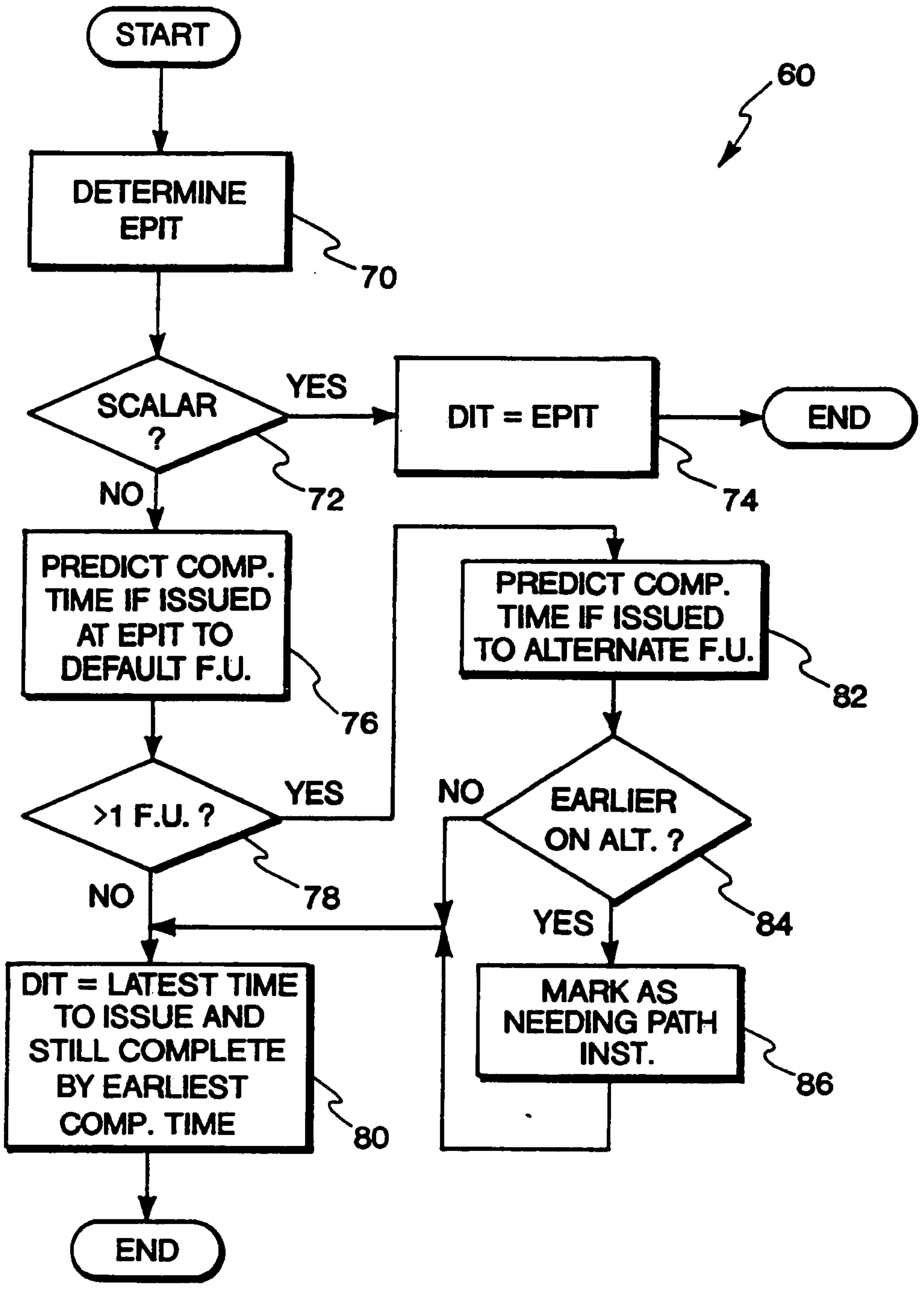

START
MOVE NODES WITH NO INCOMING EDGES TO LEADER SET
40
COMPUTE DIT FOR EACH INST. IN LEADER SET
60
MOVE NODES WHOSE DIT < SIM. TIME INTO READY SET
62
READY SET EMPTY ?
NO
YES
44
LEADER SET EMPTY ?
YES
END
NO
54
ADVANCE SIM. TIME UNTIL A NODE IN LEADER SET CAN MOVE TO READY SET
66
SCHEDULE INST. HAVING HIGHEST COST
46
REMOVE INST. AND EDGES FROM DAG
48
ADVANCE SIM. TIME BY ISSUE TIME OF THIS INST.
64
ASSIGN REGISTERS TO SCHEDULED INSTRUCTION
50
MOVE INSTRUCTIONS FROM READY TO LEADER SET IF NEW INTERLOCKS
52

Fig. 5


START
60
DETERMINE EPIT
70
SCALAR ?
YES
DIT = EPIT
END
NO
72
74
PREDICT COMP. TIME IF ISSUED AT EPIT TO DEFAULT F.U.
76
PREDICT COMP. TIME IF ISSUED TO ALTERNATE F.U.
82
>1 F.U. ?
YES
NO
78
EARLIER ON ALT. ?
NO
YES
84
DIT = LATEST TIME TO ISSUE AND STILL COMPLETE BY EARLIEST COMP. TIME
80
MARK AS NEEDING PATH INST.
86
END

INTERNATIONAL SEARCH REPORT

International Application No. PCT/US91/04073

I. CLASSIFICATION F SUBJECT MATTER (if several classification symbols apply, indicate all) 6

According to International Patent Classification (IPC) or to both NationabClassification and IPC
IPC(5): G06F 9/45,9/455,9/30
U.S. CL.: 364/200

II. FIELDS SEARCHED

Minimum Documentation Searched 7

Classification System	Classification Symbols
U.S.	364/200,900

Documentation Searched other than Minimum Documentation to the Extent that such Documents are Included in the Fields Searched 8

III. DOCUMENTS CONSIDERED TO BE RELEVANT 9

Category *	Citation of Document, 11 with indication, where appropriate, of the relevant passages 12	Relevant to Claim No. 13
A	US,A, 4,567,574 (SAADE) 28 JANUARY 1986 (see abstract)	1-6
A	US,A, 4,298,933 (SHIMOKAWA) 03 NOVEMBER 1981 (see abstract)	1-6
A	US,A, 4,399,507 (COSGROVE) 16 AUGUST 1983 (see abstract)	1-6
A	US,A, 4,179,737 (KIM) 18 DECEMBER 1979 (see abstract)	1-6

* Special categories of cited documents: 10
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier document but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
"&" document member of the same patent family

IV. CERTIFICATI N

Date of the Actual Completion of the International Search 29 AUGUST 1991	Date of Mailing of this International Search Report 03 OCT 1991
International Searching Authority ISA/US	Signature of Authorized Officer [illegible] DAVID Y. ENG